DETERMINING A RECOMMENDED SOFTWARE-STACK FOR A TARGET SOFTWARE ITEM

TECHNICAL FIELD

[0001] The present disclosure relates generally to software stacks. More specifically, but not by way of limitation, this disclosure relates to determining a recommended software-stack for a target software item.

BACKGROUND

[0002] A software stack is a set of software components configured to work in tandem to produce a result or achieve a common goal. The software components may include operating systems, architectural layers, protocols, run-time environments, databases, libraries, etc., that are stacked on top of each other in a hierarchy, such that at least some of the software components either directly or indirectly depend on others of the software components. For example, two well-known software stacks are LAMP (Linux, Apache, MySQL, and PHP/Perl/Python) and WINS (Windows Server, IIS, .NET, and SQL Server). Software stacks generally include all of the software components needed to run a particular software item (e.g., an application, service, or package), so that no additional software is needed to support the software item. The software components are typically provided together in a bundle for easy and fast installation, even though the software components are often created and maintained by different developers independently of one another.

BRIEF DESCRIPTION OF THE DRAWINGS

[0003] FIG. **1** is a block diagram of an example of a system for determining a recommended software-stack for a target software item according to some aspects of the present disclosure.

[0004] FIG. **2** is a block diagram of another example of a system for determining a recommended software-stack for a target software item according to some aspects of the present disclosure.

[0005] FIG. **3** a flow chart of an example of a process for determining a recommended software-stack for a target software item according to some aspects of the present disclosure.

DETAILED DESCRIPTION

[0006] A software item such as a software application can rely on one or more libraries or other dependencies for proper operation. There can be many versions of the software item and many versions of each of these dependencies. Additionally, at least some of the dependencies may in turn have their own dependencies that serve as indirect dependencies of the software item. There can be many versions of those indirect dependencies, and those indirect dependencies may have their own further dependencies. And so on. As a result, there is often a large number of possible combinations of a software item and its dependencies (e.g., direct and indirect dependencies) that can be used in a software stack for the software item. And some of these combinations may perform worse than others. For example, some combinations may introduce defects, assembly problems, or deployment issues that can be time consuming and difficult to identify and resolve. Given the large number of possibilities, it can be challenging to manually identify and test all of the possible combinations of a software item and its dependencies to determine the best software-stack for a software item.

[0007] Some examples of the present disclosure overcome one or more of the abovementioned problems via a system that can generate a group of software-stack candidates for a target software item and determine a respective score for each of the software-stack candidates using a scoring function. The scoring function can take into account characteristics of a computing environment in which the target software item is to be executed. The system can then select one of the software-stack candidates from the group as a recommended software-stack based on its corresponding score. In this way, the system can automatically analyze many combinations of the software item and its dependencies to determine a recommended software-stack for the software item that is the best (e.g., most optimal) for the computing environment, relative to the other software-stack candidates in the group.

[0008] As one particular example, TensorFlow is a popular Python library for building artificial intelligence or machine learning applications. An analysis of different versions of libraries used by TensorFlow 1.11.0 revealed that there are currently approximately 6.39E+27 possible combinations for a TensorFlow application stack (e.g., the TensorFlow package and possible combinations of libraries on which TensorFlow depends). And this is just for version 1.11.0 of TensorFlow. Another version of TensorFlow, version 2.0.0rc0 released on Aug. 23, 2019, currently has approximately 6.58E+35 possible combinations of libraries, which is approximately 1E+8 times more combinations than version 1.11.0. These large numbers just take libraries into account. In practice, TensorFlow runs in a computing environment with various software characteristics (e.g., native libraries and packages, cross-ecosystem dependencies, kernel versions, and driver versions) and hardware characteristics (e.g., a CPU type, GPU type, etc.), all of which add more dimensions to the search space. It would likely be impossible to manually determine and test all of the possible combinations for all of the TensorFlow versions to verify application behavior. Even trying to build and test all of the possible combinations in a more automated fashion using a computer would still be extremely time consuming and resource-intensive.

[0009] Some examples of the present disclosure can overcome one or more of the abovementioned problems via a system that can first execute a search algorithm to perform a search on a search space containing many or all possible combinations of a target software item, like TensorFlow, and its dependencies. The search algorithm can be a heuristic search algorithm, such as a Monte-Carlo tree-analysis algorithm or a temporal-difference learning algorithm; a stochastic search algorithm, such as a simulated annealing algorithm; or another type of algorithm. The search algorithm may learn or be guided over time (e.g., between iterations) to more rapidly converge towards solutions, without having to test all of the possible combinations defined in the search space. This type of searching can be referred to as a combinatorial optimization problem, in which a space is searched for a local maximum or minimum that best satisfies an objective function. By solving this combinatorial optimization problem, the system can rapidly determine a group of software-stack candidates that are most likely to yield the best results, where each of the software-stack candidates in